Sieve: Actionable Insights from Monitored Metrics in Distributed Systems

Sieve: Actionable Insights from Monitored Metrics in Distributed Systems

Jörg Thalheim[1*], Antonio Rodrigues[2*], Istemi Ekin Akkus[3], Pramod Bhatotia[1],
Ruichuan Chen[3], Bimal Viswanath[4], Lei Jiao[5*], Christof Fetzer[6]
[1]University of Edinburgh, [2]Carnegie Mellon Univ., [3]NOKIA Bell Labs, [4]University of Chicago, [5]University of Oregon, [6]TU Dresden

Abstract

Major cloud computing operators provide powerful monitoring tools to understand the current (and prior) state of the distributed systems deployed in their infrastructure. While such tools provide a detailed monitoring mechanism at scale, they also pose a significant challenge for the application developers/operators to transform the *huge space of monitored metrics* into *useful insights*. These insights are essential to build effective management tools for improving the efficiency, resiliency, and dependability of distributed systems.

This paper reports on our experience with building and deploying Sieve—a platform to derive actionable insights from monitored metrics in distributed systems. Sieve builds on two core components: a metrics reduction framework, and a metrics dependency extractor. More specifically, Sieve first reduces the dimensionality of metrics by automatically filtering out unimportant metrics by observing their signal over time. Afterwards, Sieve infers metrics dependencies between distributed components of the system using a predictive-causality model by testing for Granger Causality.

We implemented Sieve as a generic platform and deployed it for two microservices-based distributed systems: OpenStack and ShareLatex. Our experience shows that (1) Sieve can reduce the number of metrics by at least an order of magnitude ($10 - 100\times$), while preserving the statistical equivalence to the total number of monitored metrics; (2) Sieve can dramatically improve existing monitoring infrastructures by reducing the associated overheads over the entire system stack (CPU—80%, storage—90%, and network—50%); (3) Lastly, Sieve can be effective to support a wide-range of workflows in distributed systems—we showcase two such workflows: Orchestration of autoscaling, and Root Cause Analysis (RCA).

Keywords Microservices, Time series analysis

*Authors did part of the work at NOKIA Bell Labs.

1 Introduction

Most distributed systems are constantly monitored to understand their current (and prior) states. The main purpose of monitoring is to gain *actionable insights* that would enable a developer/operator to take appropriate actions to better manage the deployed system. Such insights are commonly used to manage the health and resource requirements as well as to investigate and recover from failures (root cause identification). For these reasons, monitoring is a crucial part of any distributed system deployment.

All major cloud computing operators provide a monitoring infrastructure for application developers (e.g., Amazon CloudWatch [2], Azure Monitor [12], Google StackDriver [5]). These platforms provide infrastructure to monitor a large number (hundreds or thousands) of various application-specific and system-level metrics associated with a cloud application. Although such systems feature scalable measurement and storage frameworks to conduct monitoring at scale, they leave the task of transforming the monitored metrics into usable knowledge to the developers. Unfortunately, this transformation becomes difficult with the increasing size and complexity of the application.

In this paper, we share our experience on: *How can we derive actionable insights from the monitored metrics in distributed systems?* In particular, given a large number of monitored metrics across different components (or processes) in a distributed system, we want to design a platform that can derive actionable insights from the monitored metrics. This platform could be used to support a wide-range of use cases to improve the efficiency, resiliency, and reliability of distributed systems.

In this work, we focus on microservices-based distributed systems because they have become the de-facto way to design and deploy modern day large-scale web applications [48]. The microservices architecture is an ideal candidate for our study for two reasons: First, microservices-based applications have a large number of distributed components (hundreds to thousands [45, 56]) with complex communication patterns, each component usually exporting several metrics for the purposes of debugging, performance diagnosis, and application management. Second, microservices-based applications are developed at a rapid pace: new features are being continuously integrated and deployed. Every new update may fix some existing issues, introduce new features, but can also introduce a new bug. With this rapid update schedule, keeping track of the changes in the application as a whole with effects propagating to other components becomes critical for reliability, efficiency, and management purposes.

The state-of-the-art management infrastructures either rely on ad hoc techniques or custom application-specific tools. For instance, prior work in this space has mostly focused on analyzing message-level traces (instead of monitored metrics) to generate a causal model of the application to debug performance issues [30, 42]. Alternatively, developers usually create and use custom tools to address the complexity of understanding the application as a whole. For example, Netflix developed several application-specific tools for such purposes [8, 45] by instrumenting the entire application. These approaches require either complicated instrumentation or sophisticated techniques to infer happens-before relationships (for the causal model) by analyzing message trace timestamps, making them inapplicable for broader use.

This paper presents our experience with designing and building Sieve, a system that can utilize an existing monitoring infrastructure (i.e., without changing the monitored information) to infer actionable insights for application management. Sieve takes a data-driven approach to enable better management of microservices-based applications. At its core, Sieve is composed of two key modules: (1) a metric reduction engine that reduces the dimensionality of the metric space by filtering out metrics that carry redundant information, (2) a metric dependency extractor that builds a causal

model of the application by inferring causal relationships between metrics associated with different components.

Module (1) enables Sieve to identify "relevant" metrics for a given application management task. For instance, it might be sufficient to monitor only a few metrics associated with error states of the application instead of the entire set when monitoring the health of the application. It is important to also note that reducing the metric space has implications for deployment costs: frameworks like Amazon CloudWatch use a per-metric charging model, and not identifying relevant metrics can significantly drive up the cost related to monitoring the application.

Module (2) is crucial for inferring actionable insights because it is able to automatically infer complex application dependencies. In a rapidly updating application, the ability to observe such complex dependencies and how they may change is important for keeping one's understanding of the application as a whole up-to-date. Such up-to-date information can be helpful for developers to quickly react to any problem that may arise during deployment.

We implemented Sieve as a generic platform, and deployed it with two microservices-based distributed systems: ShareLatex [22] and OpenStack [26]. Our experience shows that (1) Sieve can reduce the number of monitored metrics by an order of magnitude ($10 - 100\times$), while preserving the statistical equivalence to the total number of monitored metrics. In this way, the developers/operators can focus on the important metrics that actually matter. (2) Sieve can dramatically improve the efficiency of existing metrics monitoring infrastructures by reducing the associated overheads over the entire system stack (CPU—80%, storage—90%, and network—50%). This is especially important for systems deployed in a cloud infrastructure, where the monitoring infrastructures (e.g. AWS CloudWatch) charge customers for monitoring resources. And finally, (3) Sieve can be employed for supporting a wide-range of workflows. We showcase two such case-studies: In the first case study, we use ShareLatex [22] and show how Sieve can help developers orchestrate autoscaling of microservices-based applications. In the second case study, we use OpenStack [26] and show how developers can take advantage of Sieve's ability to infer complex dependencies across various components in microservices for Root Cause Analysis (RCA). Sieve's source code with the full experimentation setup is publicly available: https://sieve-microservices.github.io/.

2 Overview

In this section, we first present some background on microservices-based applications and our motivation to focus on them. Afterwards, we present our goals, and design overview.

2.1 Background and Motivation

Microservices-based applications consist of loosely-coupled distributed components (or processes) that communicate via well-defined interfaces. Designing and building applications in this way increases modularity, so that developers can work on different components and maintain them independently. These advantages make the microservices architecture the de facto design choice for large-scale web applications [48].

While increasing modularity, such an approach to developing software can also increase the application complexity: As the number of components increases, the interconnections between components also increases. Furthermore, each component usually exports several metrics for the purposes of debugging, performance diagnosis, and application management. Therefore, understanding the dependencies between the components and utilizing these dependencies with the exported metrics becomes a challenging task. As a result, understanding how the application performs as a whole becomes increasingly difficult.

Table 1. Metrics exposed by microservices-based applications.

Application	Number of metrics
Netflix [58]	~ 2,000,000
Quantcast [13]	~ 2,000,000
Uber [15]	~ 500,000,000
ShareLatex [22]	889
OpenStack [17, 19]	17,608

Typical microservices-based applications are composed of hundreds of components [45, 56]. Table 1 shows real-world microservices-based applications that have tens of thousands of metrics and hundreds of components. We experimented with two such applications, ShareLatex [22] and OpenStack [18], each having several thousands of metrics and order of tens of components. The metrics in these applications come from all layers of the application like hardware counters, resource usage, business metrics or application-specific metrics.

To address this data overload issue, developers of microservices-based applications usually create ad hoc tools. For example, application programmers at Netflix developed several application-specific tools for such purposes [8, 45]. These tools, however, require the application under investigation to be instrumented, so that the communication pattern between components can be established by following requests coming into the application. This kind of instrumentation requires coordination among developers of different components, which can be a limiting factor for modularity.

Major cloud computing operators also provide monitoring tools for recording all metric data from all components. For example, Amazon CloudWatch [2], Azure Monitor [12], and Google StackDriver [5]. These monitoring tools aid in visualizing and processing metric data in real-time (i.e., for performance observation) or after an issue with the application (i.e., for debugging). These tools, however, either use a few system metrics that are hand-picked by developers based on experience, or simply record all metric data for all the components.

Relying on past experience may not always be effective due to the increasing complexity of a microservices-based application. On the other hand, recording all metric data can create significant monitoring overhead in the network and storage, or in the case of running the application in a cloud infrastructure (e.g., AWS), it can incur costs due to the provider charging the customers (e.g., CloudWatch). For these reasons, it is important to understand the dependencies between the components of a microservice-based application. Ideally, this process should not be intrusive to the application. Finally, it should help the developers to identify and minimize the critical components and metrics to monitor.

2.2 Design Goals

While designing Sieve, we set the following goals.

- **Generic:** Many tools for distributed systems have specific goals, including performance debugging, root cause analysis and orchestration. Most of the time, these tools are custom-built for the application in consideration and target a certain

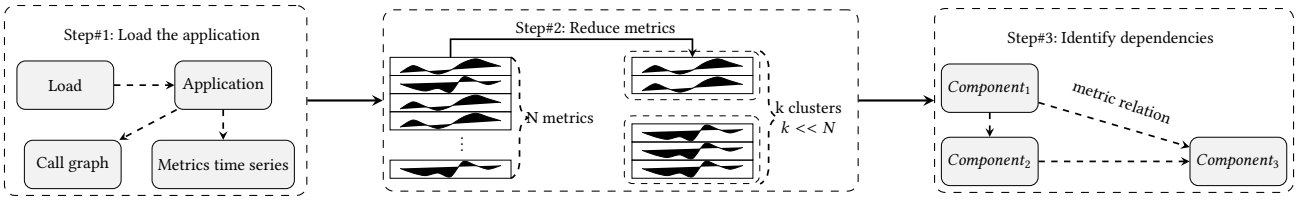

Figure 1. High level steps of Sieve.

goal. Our goal is to design a generic platform that can be used for a wide-range of workflows.

- **Automatic:** The sheer number of metrics prohibits manual inspection. On the other hand, designing a generic system to help developers in many use cases might require manually adjusting some parameters for each use case. Our tool should be as automated as possible while reducing the number of metrics and extracting their relationships. However, we leave the utilization of our platform's output to the developers, who may have different goals.
- **Efficient:** Our platform's operation should be as efficient as possible. Minimizing analysis time becomes important when considering distributed systems, such as microservices-based applications.

2.3 Sieve Overview

The underlying intuition behind Sieve is two-fold: Firstly, in the metric dimension, some metrics of a component may behave with similar patterns as other metrics of that component. Secondly, in the component dimension, there are dependencies between components. As a result, monitoring all metrics of all components at runtime may be unnecessary and inefficient (as components are not independent).

In this paper, we present Sieve to reduce this complexity by systematically analyzing the application to filter collected metrics and to build a dependency graph across components. To showcase the generality of this dependency graph and its benefits, we then utilize Sieve to orchestrate autoscaling of the ShareLatex [22] application—an online collaboration tool, and to perform Root Cause Analysis (RCA) in OpenStack [26]—a cloud management software (§4).

At a high level, Sieve's design follows three steps as shown in Figure 1.

Step #1: Load the application. Sieve uses an application-specific load generator to stress the application under investigation. This load generator can be provided by the application developers. For example, OpenStack already uses a load generator named Rally [20]. During the load, Sieve records the communications among components to obtain a *call graph*. This recording does not require any modifications to the application code. In addition, Sieve records all exposed *metrics* by all components. Note that this recording only happens during the creation of the call graph and not during runtime.

Step #2: Reduce metrics. After collecting the metrics, Sieve analyzes each component and organizes its metrics into fewer groups via clustering, so that similar-behaving metrics are clustered together. After clustering, Sieve picks a representative metric from each cluster. These representative metrics as well as their clusters in a sense characterize each component.

Step #3: Identify dependencies. In this step, Sieve explores the possibilities of one component's representative metrics affecting another component's metrics using a pairwise comparison method: each representative metric of one component is compared with each representative metric of another component. Sieve uses the call graph obtained in Step 1 to choose the components to be compared (i.e., components directly communicating) and the representative metrics determined in Step 2. As a result, the search space is significantly reduced compared to the naïve approach of comparing all components with every other component using all metrics.

If Sieve determines that there is a relationship between a metric of one component and another metric of another component, a dependency edge between these components is created using the corresponding metrics. The direction of the edge depends on which component is affecting the other.

3 Design

In this section, we detail the three steps of Sieve.

3.1 Load the Application

For our systematic analysis, we first run the application under various load conditions. This loading serves two purposes: First, the load exposes a number of metrics from the application as well as the infrastructure it runs on. These metrics are then used to identify potential relationships across components. Second, the load also enables us to obtain a call graph, so that we can identify the components that communicate with each other. The call graph is later used to reduce the amount of computation required to identify the inter-component relationships (§3.3). The load test is intended to be run in an offline step and not in production.

Obtaining metrics. During the load of the application, we record metrics as time series. There are two types of metrics that we can leverage for our analysis: First, there are system metrics that are obtained from the underlying operating system. These metrics report the resource usage of a microservice component, and are usually related to the hardware resources on a host. Examples include usages in CPU, memory, network and disk I/O.

Second, there are application-level metrics. Application developers often add application-specific metrics (e.g., number of active users, response time of a request in a component). Commonly-used components (e.g., databases, load balancers) and certain language runtimes (e.g., Java) may provide statistics about specific operations (e.g., query times, request counts, duration of garbage collection).

Obtaining the call graph. Generally speaking, applications using a microservices architecture communicate via well-defined interfaces similar to remote procedure calls. We model these communications between the components as a directed graph, where the vertices represent the microservice components and the edges point from the caller to the callee providing the service.

By knowing which components communicate directly, we can reduce the number of component pairs we need to check to see whether they have a relation (see Section 3.3). Although it is possible to manually track this information for smaller-sized applications, this process becomes quickly difficult and error-prone with increasing number of components.

There are several ways to understand which microservice components are communicating with each other. One can instrument the application, so that each request can be traced from the point it enters the application to the point where the response is returned to the user. Dapper [81] from Google and Atlas [45, 58] from Netflix rely on instrumenting their RPC middleware to trace requests.

Another method to obtain communicating components is to monitor network traffic between hosts running those components using a tool like *tcpdump*. After obtaining the traffic, one can map the exchanged packets to the components via their source/destination addresses. This method can produce communicating component pairs by parsing all network packets, adding significant computational overhead and increasing the analysis time. Furthermore, it is possible that many microservice components are deployed onto the same host (e.g., using containers), making the packet parsing difficult due to network address translation on the host machine.

One can also observe system calls related to network operations via APIs such as *ptrace()* [28]. However, this approach adds a lot of context switches between the tracer and component under observation.

Sieve employs *sysdig* to obtain the communicating pairs. *sysdig*[23] is a recent project providing a new method to observe system calls in a more efficient way. Utilizing a kernel module, *sysdig* provides system calls as an event stream to a user application. The event stream also contains information about the monitored processes, so that network calls can be mapped to microservice components, even if they are running in containers. Furthermore, it enables extraction of the communication peer via user-defined filters. Employing *sysdig*, we avoid the shortcomings of the above approaches: 1) We do not need to instrument the application, which makes our system more generally applicable, 2) We add little overhead to obtain the call graph of an application for our analysis (see Section 6.1.3).

3.2 Reduce Metrics

The primary goal of exporting metrics is to understand the performance of applications, orchestrating them and debugging them. While the metrics exported by the application developers or commonly-used microservice components may be useful for these purposes, it is often the case that the developers have little idea regarding which ones are going to be most useful. Developers from different backgrounds may have different opinions: a developer specializing in network communications may deem network I/O as the most important metric to consider, whereas a developer with a background on algorithms may find CPU usage more valuable. As a result of these varying opinions, often times many metrics are exported.

While it may look like there is no harm in exporting as much information as possible about the application, it can create problems. Manually investigating the obtained metrics from a large number of components becomes increasingly difficult with the increasing number of metrics and components [35]. This complexity reflects on the decisions that are needed to control and maintain the application. In addition, the overhead associated with the collection and storage of these metrics can quickly create problems. In fact, Amazon CloudWatch [2] charges its customers for the reporting of the metrics they export. As a result, the more metrics an application has to export, the bigger the cost the developers would have to bear.

One observation we make is that some metrics strongly correlate with each other and it might not be necessary to consider all of them when making decisions about the control of the application. For example, some application metrics might be strongly correlated with each other due to the redundancy in choosing which metrics to export by the developers. It is also possible that different subsystems in the same component report similar information (e.g., overall memory vs. heap usage of a process). In addition, some system metrics may offer clues regarding the application's state: increased network I/O may indicate an increase in the number of requests.

The direct outcome of this observation is that it should be possible to reduce the dimensionality of the metrics the developers have to consider. As such, the procedure to enable this reduction should happen with minimal user effort and scale with increased numbers of metrics.

To achieve these requirements, Sieve uses a clustering approach named k-Shape [73] with a pre-filtering step. While other approaches such as principal component analysis (PCA) [49] and random projections [72] can also be used for dimensionality reduction, these approaches either produce results that are not easily interpreted by developers (i.e., PCA) or sacrifice accuracy to achieve performance and have stability issues producing different results across runs (i.e., random projections). On the other hand, clustering results can be visually inspected by developers, who can also use any application-level knowledge to validate their correctness. Additionally, clustering can also uncover hidden relationships which might not have been obvious.

Filtering unvarying metrics. Before we use k-Shape, we first filter metrics with constant trend or low variance ($var \leq 0.002$). These metrics cannot provide any new information regarding the relationships across components, because they are not changing according to the load applied to the application. Removing these metrics also enables us to improve the clustering results.

k-Shape clustering. k-Shape is a recent clustering algorithm that scales linearly with the number of metrics. It uses a novel distance metric called *shape-based distance* (SBD). SBD is based on a normalized form of cross correlation (NCC) [73]. Cross correlation is calculated using Fast Fourier Transformation and normalized using the geometric mean of the autocorrelation of each individual metric's time series. Given two time series vectors, $\vec{x}$ and $\vec{y}$, SBD will take the position w, when sliding $\vec{x}$ over $\vec{y}$, where the normalized cross correlation maximizes.

$$SBD(\vec{x}, \vec{y}) = 1 - max_w \left(NCC_w(\vec{x}, \vec{y})\right) \quad (1)$$

Because k-Shape uses a distance metric based on the shape of the investigated time series, it can detect similarities in two time series, even if one lags the other in the time dimension. This feature is important to determine relationships across components in microservices-based applications because a change in one metric in one component may not reflect on another component's metrics immediately (e.g., due to the network delay of calls between components).

Additionally, k-Shape is robust against distortion in amplitude because data is normalized via z-normalization ($z = \frac{x-\mu}{\sigma}$) before

being processed. This feature is especially important because different metrics may have different units and thus, may not be directly comparable.

k-Shape works by initially assigning time series to clusters randomly. In every iteration, it computes new cluster centroids according to SBD with the assigned time series. These centroids are then used to update the assignment for the next iteration until the clusters converge (i.e., the assignments do not change).

We make three adjustments to employ *k*-Shape in Sieve. First, we preprocess the collected time series to be compatible with *k*-Shape. *k*-Shape expects the observations to be equidistantly distributed in the time domain. However, during the load of the application, timeouts or lost packets can cause gaps between the measurements.

To reconstruct missing data, we use spline interpolation of the third order (cubic). A spline is defined piecewise by polynomial functions. Compared to other methods such as averages of previous values or linear interpolation, spline interpolation provides a higher degree of smoothness. It therefore introduces less distortion to the characteristics of a time-series [66]. Additionally, monitoring systems retrieve metrics at different points in time and need to be discretized to match each other. In order to increase the matching accuracy, we discretize using 500*ms* instead of the original 2*s* used in the original *k*-Shape paper [73].

Our second adjustment is to change the initial assignments of metric time series to clusters. To increase clustering performance and reduce the convergence overhead, we pre-cluster metrics according to their name similarity (e.g., Jaro distance [60]) and use these clusters as the initial assignment instead of the default random assignment. This adjustment is reasonable given that many developers use naming conventions when exporting metrics relating to the same component or resource in question (e.g., "cpu_usage", "cpu_usage_percentile"). The number of iterations to converge should decrease compared to the random assignment, because similar names indicate similar metrics. Note that this adjustment is only for performance reasons; the convergence of the *k*-Shape clustering does not require any knowledge of the variable names and would not be affected even with a random initial assignment.

During the clustering process, *k*-Shape requires the number of clusters to be previously determined. In an application with several components, each of which having various number of metrics, pre-determining the ideal number of clusters may not be straightforward. Our final adjustment is to overcome this limitation: we iteratively vary the number of clusters used by *k*-Shape and pick the number that gives the best *silhouette value* [78], which is a technique to determine the quality of the clusters. The silhouette value is -1 when the assignment is wrong and 1 when it is a perfect assignment [29]. We use the SBD as a distance measure in the silhouette computation.

In practice, experimenting with a small number of clusters is sufficient. For our applications, seven clusters per component was sufficient, where each component had up to 300 metrics.

Representative metrics. After the clustering, each microservice component will have one or more clusters of metrics. The number of clusters will most likely be much smaller than the number of metrics belonging to that component. Once these clusters are obtained, Sieve picks one representative metric from each cluster. To pick the representative metric from each cluster, Sieve determines the SBD between each metric and the corresponding centroid of the cluster. The metric with the lowest metric is chosen as the representative metric for this cluster.

The high-level idea is that the behavior of the cluster will match this representative metric; otherwise, the rest of the metrics in the cluster would not have been in the same cluster as this metric. The set of representative metrics of a component can then be used to describe a microservice component's behavior. These representative metrics are then used in conjunction with the call graph obtained in Section 3.1 to identify and understand the relationships across components.

3.3 Identify Dependencies

To better understand an application, we need to find dependencies across its components. A naïve way of accomplishing this goal would be to compare all components with each other using all possible metrics. One can clearly see that with the increasing number of components and metrics, this would not yield an effective solution.

In the previous section, we described how one can reduce the number of metrics one has to consider in this pairwise comparison by clustering and obtaining the representative metrics of each component. Still, comparing all pairs of components using this reduced set of metrics may be inefficient and redundant considering the number of components in a typical microservices-based application (e.g., tens or hundreds).

Sieve uses the call graph obtained in Section 3.1 to reduce the number of components that need to be investigated in a pairwise fashion. For each component, we do pairwise comparisons using each representative metric of its clusters with each of its neighbouring components (i.e., callees) and their representative metrics.

Sieve utilizes Granger Causality tests [54] in this pairwise comparison. Granger Causality tests are useful in determining whether a time series can be useful in predicting another time series: In a microservices-based application, the component interactions closely follow the path a request takes inside the application. As a result, these interactions can be predictive of the changes in the metrics of the components in the path. Granger Causality tests offer a statistical approach in understanding the relationships across these components. Informally, Granger Causality is defined as follows. If a metric X is Granger-causing another metric Y, then we can predict Y better by using the history of both X and Y compared to only using the history of Y [51].

To utilize Granger Causality tests in Sieve, we built two linear models using the ordinary least-square method [32]. First, we compare each metric X_t with another metric Y_t. Second, we compare each metric X_t with the time-lagged version of the other metric Y_t: Y_{t-Lag}. Covering the cases with a time lag is important because the load in one component may not be reflected on another component until the second component receives API calls and starts processing them.

Sieve utilizes short delays to build the time-lagged versions of metrics. The reason is that microservices-based applications typically run in the same data center and their components communicate over a LAN, where typical round-trip times are in the order of milliseconds. Sieve uses a conservative delay of 500*ms* for unforeseen delays.

To apply the Granger Causality tests and check whether the past values of metric X can predict the future values of metric Y, both models are compared via the F-test [67]. The null hypothesis (i.e.,

X does not granger-cause Y) is rejected if the p-value is below a critical value.

However, one has to consider various properties of the time series. For example, the F-test requires the time series to be normally distributed. The load generation used in Section 3.1 can be adjusted to accommodate this requirement. Also, the F-test might find spurious regressions when non-stationary time series are included [53]. Non-stationary time series (e.g., monotonically increasing counters for CPU and network interfaces) can be found using the Augmented Dickey-Fuller test [55]. For these time series, the first difference is taken and then used in the Granger Causality tests. Although longer trends may be lost due to the first difference, accumulating metrics such as counters do not present interesting relationships for our purposes.

After applying the Granger Causality test to each component's representative metrics with its neighbouring component's representative metrics, we obtain a graph. In this graph, we draw an edge between microservice components, if one metric in one component Granger-causes another metric in a neighbouring component. This edge represents the dependency between these two components and its direction is determined by Granger causality.

While Granger Causality tests are useful in determining predictive causality across microservice components, it has some limitations that we need to consider. For example, it does not cover instantaneous relationships between two variables. More importantly, it might reveal spurious relationships, if important variables are missing in the system: if both X and Y depend on a third variable Z that is not considered, any relationship found between X and Y may not be useful. Fortunately, an indicator of such a situation is that both metrics will Granger-cause each other (i.e., a bidirectional edge in the graph). Sieve filters these edges out.

4 Applications

In this section, we describe two use cases to demonstrate Sieve's ability to handle different workflows. In particular, using Sieve's base design, we implemented 1) an orchestration engine for autoscaling and applied it to ShareLatex [22], and 2) a root cause analysis (RCA) engine and applied it to OpenStack [18].

4.1 Orchestration of Autoscaling

For the autoscaling case study, we used ShareLatex [22]—a popular collaborative LaTeX editor. ShareLatex is structured as a microservices-based application, delegating tasks to multiple well-defined components that include a KV-store, load balancer, two databases and 11 node.js based components.

Sieve's pairwise investigation of representative metrics of components produces the dependencies across components. By leveraging this dependency graph, our autoscaling engine helps developers to make more informed decisions regarding which components and metrics are more critical to monitor. As a result, developers can generate *scaling rules* with the goal of adjusting the number of active component instances, depending on real-time workload.

More specifically, we use Sieve's dependency graph and extract (1) *guiding metrics* (i.e., metrics to use in a scaling rule), (2) *scaling actions* (i.e., actions associated with reacting to varying loads by increasing/decreasing the number of instances subject to minimum/maximum thresholds), and (3) *scaling conditions* (i.e., conditions based on a guiding metric triggering the corresponding scaling action). Below, we explain how we use Sieve to generate a scaling rule:

#1: Metric. We pick a metric m that appears the most in Granger Causality relations between components.

#2: Scaling actions. In our case study, we restrict scaling actions to scale in/out actions, with increments/decrements of a single component instance ($+/-1$).

#3: Conditions. The scale in/out thresholds are defined from the values of m according to a Service Level Agreement (SLA) condition. For ShareLatex, such an SLA condition can be to keep 90% of all request latencies below 1000ms. The thresholds for m are iteratively refined during the application loading phase.

Table 2. Description of dependency graph differences considered by the root cause analysis engine.

Scoping level	Differences of interest
Component metrics	Present in F version, not in C (*new*) Present in C version, not in F (*discarded*)
Clusters	Cluster includes new/discarded metrics
Dep. graph edges	New/discarded edge between similar clusters Different time-lag between similar clusters Includes clusters w/ new/discarded metrics

4.2 Root Cause Analysis

For the root cause analysis (RCA) case study, we used OpenStack [18, 26], a popular open-source cloud management software. OpenStack is structured as a microservices-based application with a typical deployment of ~10 (or more) individual components, each often divided into multiple sub-components [80]. Due to its scale and complexity, OpenStack is susceptible to faults and performance issues, often introduced by updates to its codebase.

In microservices-based applications such as Openstack, components can be updated quite often [59], and such updates can affect other application components. If relationships between components are complex, such effects may not be easily foreseeable, even when inter-component interfaces are unchanged (e.g., if the density of inter-component relationships is high or if the activation of relationships is selective depending on the component's state and inputs). Sieve's dependency graph can be used to understand the update's overall effect on the application: changing dependency graphs can indicate potential problems introduced by an update. By identifying such changes, Sieve can help developers identify the root cause of the problem.

Our RCA engine leverages Sieve to generate a list of possible root causes of an anomaly in the monitored application. More specifically, the RCA engine compares the dependency graphs of two different versions of an application: (1) a *correct* version; and (2) a *faulty* version. Similarly to [61, 63], we assume that the system anomaly (but not its *cause*) has been observed and the correct and faulty versions have been identified. The result of this comparison is a list of {*component*, *metric list*} pairs: the *component* item points to a component as a possible source for the issue, whereas the *metric list* shows the metrics in that component potentially related to the issue, providing a more fine-grained view. With the help of this list, developers can reduce the complexity of their search for the root cause.

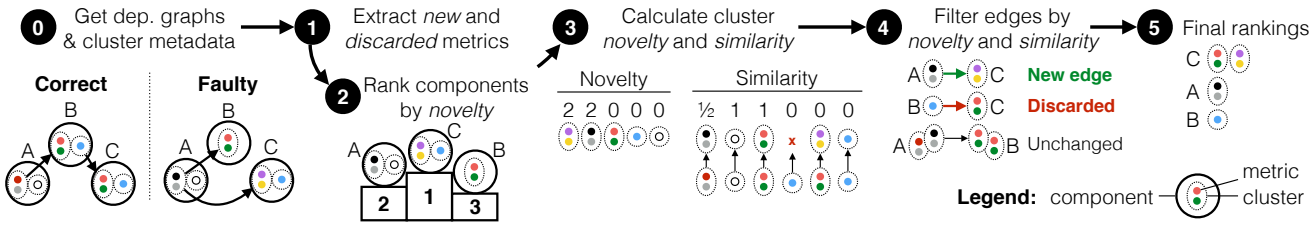

Figure 2. Sieve's root cause analysis methodology.

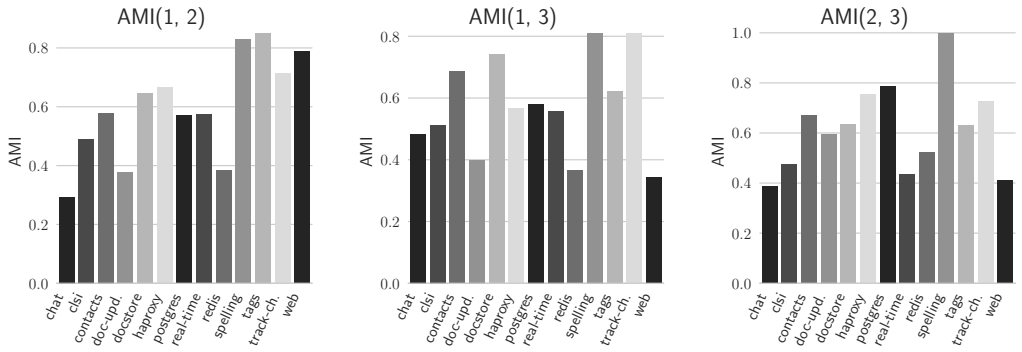

Figure 3. Pairwise adjusted mutual information (AMI) scores between 3 measurements.

Figure 2 shows the five steps involved in the comparison. At each step, we extract and analyze Sieve's outputs at three different granularity levels: *metrics*, *clusters*, and *dependency graph edges*. The levels and corresponding differences of interest are described in Table 2. We describe the steps in more detail below.

#1: Metric analysis. This step analyzes the presence or absence of metrics between C and F versions. If a metric m is present in both C and F, it intuitively represents the maintenance of healthy behavior associated with m. As such, these metrics are filtered out of this step. Conversely, the appearance of a new metric (or the disappearance of a previously existing metric) between versions is likely to be related with the anomaly.

#2: Component rankings. In this step, we use the results of step 1 to rank components according to their *novelty score* (i.e., total number of new or discarded metrics), producing an initial group of interesting components for RCA.

#3: Cluster analysis: novelty & similarity. Clusters aggregate component metrics which exhibit similar behavior over time. The clusters with new or discarded metrics should be more interesting for RCA compared to the unchanged clusters of that component (with some exceptions, explained below). For a given component, we compute the novelty scores of its clusters as the sum of the number of new and discarded metrics, and produce a list of {*component*, *metric list*} pairs, where the metric list considers metrics from the clusters with higher novelty scores.

In addition, we track the similarity of a component's clusters between C and F versions (or vice-versa). This is done to identify two events: (1) appearance (or disappearance) of edges between versions; and (2) attribute changes in relationships maintained between C and F versions (e.g., a change in Granger causality time lag). An edge between clusters x and y (belonging to components A and B, respectively) is said to be 'maintained between versions' if their respective metric compositions do not change significantly between C and F versions, i.e. if $S(\mathcal{M}^A_{x,\mathrm{C}}) \approx S(\mathcal{M}^A_{x',\mathrm{F}})$ **and** $S(\mathcal{M}^B_{y,\mathrm{C}}) \approx S(\mathcal{M}^B_{y',\mathrm{F}})$. $\mathcal{M}^A_{x,\mathrm{C}}$ and $\mathcal{M}^A_{x',\mathrm{F}}$ are the metric compositions of clusters x and x' of component A, in the C and F versions, respectively. S is some measure of cluster similarity (defined below). Both events – (1) and (2) – can be an indication of an anomaly, because one would expect edges between clusters with high similarity to be maintained between versions.

We compute the *cluster similarity score*, S, according to a modified form of the Jaccard similarity coefficient

$$S = \frac{|\mathcal{M}^A_{i,\mathrm{C}} \cap \mathcal{M}^A_{j,\mathrm{F}}|}{|\mathcal{M}^A_{i,\mathrm{C}}|} \tag{2}$$

To eliminate the penalty imposed by new metrics added to the faulty cluster, we only consider the contents of the correct cluster in the denominator (instead of the union of $\mathcal{M}^A_{i,\mathrm{C}}$ and $\mathcal{M}^A_{j,\mathrm{F}}$).

#4: Edge filtering. To further reduce the list of {*component*, *metric list*} pairs, we examine the relationships between components and clusters identified in steps 2 and 3. We identify three events:

1. Edges involving (at least) one cluster with a high novelty score
2. Appearance or disappearance of edges between clusters with high similarity
3. Changes in time lag in edges between clusters with high similarity

Event 1 isolates metrics related to edges which include at least one 'novel' cluster. Events 2 and 3 isolate clusters which are maintained between C and F versions, but become interesting for RCA due to a change in their relationship. Novelty and similarity scores are computed as in step 3. We define thresholds for 'high' novelty and similarity scores.

#5: Final rankings. We present a final list of {*component*, *metric list*} pairs. The list is ordered by component, following the rank given in step 2. The *metric list* items include the metrics identified at steps 3 and 4.

5 Implementation

We next describe the implementation details of Sieve. Our system implementation, including used software versions, is published at https://sieve-microservices.github.io. For load generation, Sieve requires an application-specific load generator. We experimented with two microservices-based applications: ShareLatex [22] and OpenStack [18, 26]. For ShareLatex, we developed our own load generator using Locust [10], a Python-based distributed load generation tool to simulate virtual users in the application (1,041 LoC). For OpenStack, we used Rally [20], the official benchmark suite from OpenStack.

For metric collection, Sieve uses Telegraf [24] to collect application/system metrics and stores them in InfluxDB [7]. Telegraf seamlessly integrates with InfluxDB, supports metrics of commonly-used components (e.g., Docker, RabbitMQ, memcached) and can run custom scripts for collection of additional metrics exposed by application APIs (e.g., [19]). With this setup, Sieve can store any time-series metrics exposed by microservice components.

For the call graph extraction, Sieve leverages sysdig call tracer [23] to obtain which microservice components communicate with each other. We wrote custom scripts to record network system calls with source and destination IP addresses on every machine hosting the components (457 LoC). These IP addresses are then mapped to the components using the cluster manager's service discovery mechanism.

We implemented Sieve's data analytics techniques in Python (2243 LoC) including metric filtering, clustering based on k-Shape, and Granger Causality. The analysis can also be distributed across multiple machines for scalability.

Lastly, we also implemented two case studies based on the Sieve infrastructure: autoscaling in ShareLatex (720 LoC) and RCA in OpenStack (507 LoC). For our autoscaling engine, we employed Kapacitor [9] to stream metrics from InfluxDB in real-time and to install our scaling rules using its user-defined functions. For the RCA engine, we implemented two modules in Python: one module extracts metric clustering data (125 LoC) and the other module (382 LoC) compares clustering data and dependency graphs.

6 Evaluation

Our evaluation answers the following questions:

1. How effective is the general Sieve framework? (§6.1)
2. How effective is Sieve for autoscaling? (§6.2)
3. How effective is Sieve for root cause analysis? (§6.3)

6.1 Sieve Evaluation

Before we evaluate Sieve with the case studies, we evaluate Sieve's general properties: *(a)* the robustness of clustering; *(b)* the effectiveness of metric reduction; and *(c)* the monitoring overhead incurred by Sieve's infrastructure.

Experimental setup. We ran our measurements on a 10 node cluster, every node with a 4-core Xeon E5405 processor, 8 GB DDR2-RAM and a 500GB HDD. For the general experiments, we loaded ShareLatex using Sieve five times with random workloads. The random workloads also help to validate whether the model stays consistent, if no assumption about the workload is made.

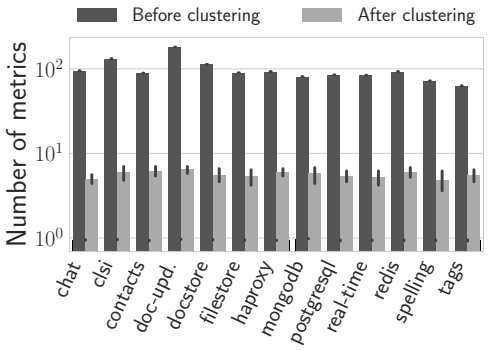

Figure 4. Average no. of metrics after Sieve's reduction.

6.1.1 Robustness

We focus on two aspects to evaluate Sieve's robustness. First, we investigate the *consistency* of clustering across different runs. Second, we try to *validate* whether the metrics in a cluster indeed belong together.

Consistency. To evaluate consistency, we compare cluster assignments produced in different measurements. A common metric to compare cluster assignments is Adjusted Mutual Information (AMI) score [85]. AMI is normalized against a random assignment and ranges from zero to one: If AMI is equal to one, both clusters match perfectly. Random assignments will be close to zero.

Figure 3 shows the AMI of cluster assignments for individual components for three independent measurements. To reduce the selection bias we apply randomized workload in a controlled environment. As a result, they should constitute a worst-case performance for the clustering. Our measurements show that the average AMI is 0.597, which is better than random assignments. Based on these measurements, we conclude the clusterings are consistent.

Validity. To evaluate the validity of the clusters, we choose three criteria: *(1)* Is there a common visible pattern between metrics in one cluster? *(2)* Do metrics in a cluster belong together assuming application knowledge? *(3)* Are the shape-based distances between metrics and their cluster centroid below a threshold (i.e., 0.3)?

We choose three clusters with different Silhouette scores (high, medium, low). According to the above criteria, we conclude the clustering algorithm can determine similar metrics. For example, application metrics such as HTTP request times and corresponding database queries are clustered together. Similar to consistency, higher Silhouette scores indicate that the clusters are more meaningful and potentially more useful for the developers. We omit the details for brevity.

6.1.2 Effectiveness

The purpose of clustering is to reduce the number of metrics exposed by the system without losing much information about the system behavior. To evaluate how effective our clustering is in reducing the number of metrics, we compare the results of the clustering with the actual number of metrics in the application. We identified 889 unique metrics within ShareLatex, meaning that an operator would have to understand and filter these metrics. Sieve's clustering reduces this number to 65 (averaged across five runs). Figure 4 shows the reduction in the number of metrics for the individual components in ShareLatex. Note that this measurement is

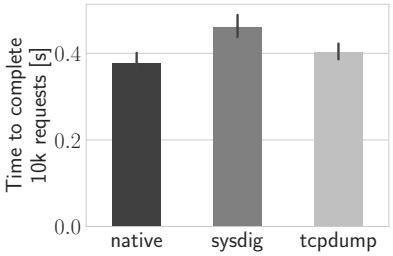

Figure 5. Completion time for HTTP requests when using tcpdump, sysdig or native (i.e., no monitoring).

Table 3. InfluxDB overhead before Sieve's reduction of metrics.

Metric	Before	After	Reduction
CPU time [s]	0.45G	0.085G	81.2 %
DB size [KB]	588.8	36.0	93.8 %
Network in [MB]	11.1	2.3	79.3 %
Network out [KB]	15.1	7.4	50.7 %

with high Silhouette scores for the clusters, which implies that the metrics reduction does not affect the quality of the clusters.

6.1.3 Monitoring Overhead

We evaluate Sieve's overhead based on two aspects. First, we compare different techniques for obtaining the call graph and show how our approach fairs. Second, we investigate the overhead incurred by the application by comparing the monitoring overhead with and without using Sieve.

Overheads. To measure the monitoring overhead during the loading stage, we run an experiment with 10K HTTP requests for a small static file using Apache Benchmark [11] on an Nginx web server [14]. Because the computational overhead for serving such a file is low, this experiment shows the worst-case performance for sysdig and tcpdump. Figure 5 shows the time it takes to complete the experiment. While tcpdump incurs a lower overhead than sysdig (i.e., 7% vs. 22%), it provides less context regarding the component generating the request and requires more knowledge about the network topology to obtain the call graph. sysdig provides all this information without much additional overhead.

Gains. To show the gains during the runtime of the application after using Sieve, we compare the computation, storage and network usage for the metrics collected during the five measurements. We store all collected metrics in InfluxDB and measure the respective resource usage. We then repeat the same process using the metrics found by Sieve; thus, simulating a run with the reduced metrics. Table 3 shows the relative usage of the respective resources with Sieve. Sieve reduces the monitoring overhead for computation, storage and network by 80%, 90% and 50%, respectively.

6.2 Case-study #1: Autoscaling

We next evaluate the effectiveness of Sieve for the orchestration of autoscaling in microservices.

Experimental setup. For the autoscaling case study, we used ShareLatex [22] (as described in §4.1). We used 12 t2.large VM-Instances on Amazon EC2 with 2 vCPUs, 8GB RAM and 20 GB Amazon EBS storage. This number of instances were sufficient to stress-test all components of the application. The VM instances were allocated statically during experiments as Docker containers. We created a Docker image for each ShareLatex component and used Rancher [21] as the cluster manager to deploy our containers across different hosts.

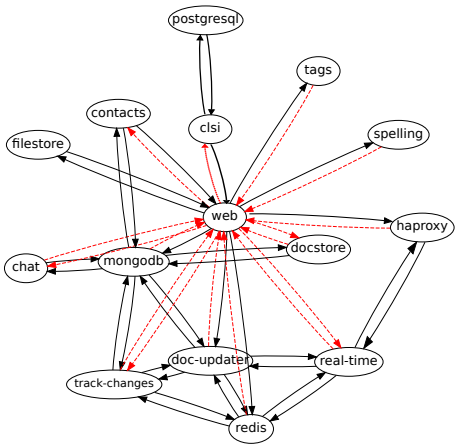

Figure 6. Relations between components based on Granger Causality in ShareLatex. The dashed lines denote relationships with metric http-requests_Project_id_GET_mean.

Dataset. We used a HTTP trace sample from soccer world cup 1998 [6] for an hour long trace. Note that the access pattern and requested resources in the world cup trace differs from the ShareLatex application. However, we used the trace to map traffic patterns for our application to generate a realistic spike workload. In particular, sessions in the HTTP trace were identified by using the client IP. Afterwards, we enqueued the sessions based on their timestamp, where a virtual user was spawned for the duration of each session and then stopped.

Results. We chose an SLA condition, such that $90th$ percentile of all request latencies should be below 1000ms. Traditional tools, such as Amazon AWS Auto Scaling [1], often use the CPU usage as the default metric to trigger autoscaling. Sieve identified an application metric named *http-requests_Project_id_GET_mean* (Figure 6) as a better metric for autoscaling than CPU usage.

To calculate the threshold values to trigger autoscaling, we used a 5-minute sample from the peak load of our HTTP trace and iteratively refined the values to stay within the SLA condition. As a result, we found that the trigger thresholds for scaling up and down while using the CPU usage metric should be 21% and 1%, respectively. Similarly, for *http-requests_Project_id_GET_mean*, the thresholds for scaling up and down should be 1400ms and 1120ms, respectively.

After installing the scaling actions, we ran our one-hour trace. Table 4 shows the comparison when using the CPU usage and *http-requests_Project_id_GET_mean* for the scaling triggers. When Sieve's selection of metric was used for autoscaling triggers, the average CPU usage of each component was increased. There were also fewer SLA violations and scaling actions.

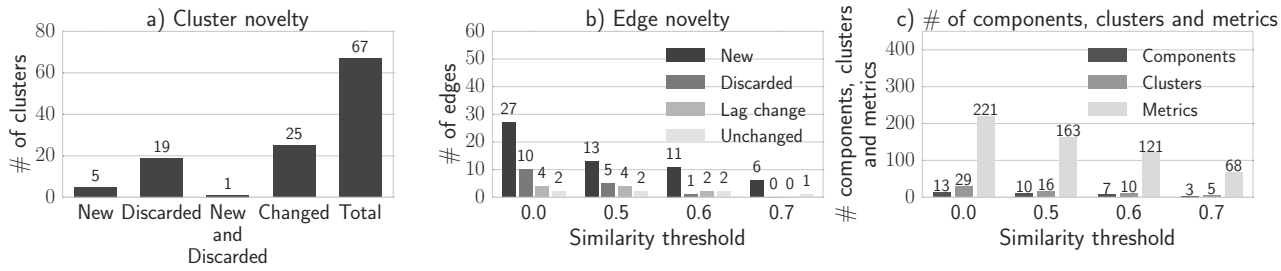

Figure 7. (a) Cluster novelty score. **(b)** Edge novelty score. **(c)** No. of components & clusters after edge filtering w/ varying thresholds.

Table 4. Comparison between a traditional metric (CPU usage) and Sieve's selection when used as autoscaling triggers.

Metric	CPU usage	Sieve	Difference [%]
Mean CPU usage per component	5.98	9.26	+54.82
SLA violations (out of 1400 samples)	188	70	-62.77
Number of scaling actions	32	21	-34.38

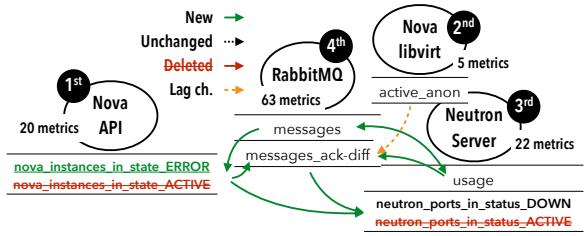

Figure 8. Final edge differences for RCA evaluation between top 5 components of Table 5 with similarity threshold of 0.50.

6.3 Case-study #2: Root Cause Analysis

To evaluate the applicability of Sieve to root cause analysis, we reproduce a representative OpenStack anomaly, Launchpad bug #1533942 [27]. We selected this issue because it has well-documented root causes, providing an appropriate ground truth, and allowing for the identification of 'correct' and 'faulty' code versions. We compare the documented root causes to the lists of root causes produced by our RCA engine. A similar failure is used as a representative case in prior work [52, 80]. Due to space constraints, we refer the analysis of other representative bugs to an extended version of the article available at [84].

Bug description: Failure to launch a VM. The bug manifests itself as follows: when launching a new VM instance using the command line interface, one gets the error message *'No valid host was found. There are not enough hosts available.'* despite the availability of compute nodes. Without any other directly observable output, the instance falls into 'ERROR' state and fails.

Root cause. The failure is caused by the crash of an agent in the Neutron component, namely the Open vSwitch agent. The Open vSwitch agent is responsible for setting up and managing virtual networking for VM instances. The ultimate cause is traced to a configuration error in OpenStack Kolla's deployment scripts [27].

Experimental setup. We deployed OpenStack components as containerized microservices using Kolla [26]. We configured Kolla to deploy 7 main OpenStack components (e.g., Nova, Neutron, Keystone, Glance, Ceilometer) along with several auxiliary components

Table 5. OpenStack components, sorted by the number of novel metrics between correct (C) and faulty (F) versions.

Component	Changed (New/Discarded)	Total (per component)	Final ranking
Nova API	29 (7/22)	59	1
Nova libvirt	21 (0/21)	39	2
Nova scheduler	14 (7/7)	30	-
Neutron server	12 (2/10)	42	3
RabbitMQ	11 (5/6)	57	4
Neutron L3 agent	7 (0/7)	39	5
Nova novncproxy	7 (0/7)	12	-
Glance API	5 (0/5)	27	6
Neutron DHCP ag.	4 (0/4)	35	7
Nova compute	3 (0/3)	41	8
Glance registry	3 (0/3)	23	9
Haproxy	2 (1/1)	14	10
Nova conductor	2 (0/2)	29	-
Other 3 components	0 (0/0)	59	-
Totals	**113 (22/91)**	**508**	-

(e.g., RabbitMQ, memcached) for a total of 47 microservices. We use OpenStack's telemetry component (Ceilometer) to expose relevant OpenStack-related metrics and extract them via Telegraf. The infrastructure consists of two m4.xlarge Amazon EC2 VM instances to run OpenStack components (16 vCPUs, 64 GB RAM and 20 GB Amazon EBS storage) and three t2.medium VM instances (2 vCPUs, 4GB RAM and 20 GB EBS storage) for the supporting components (measurement, database and deployment).

Results. We expect the RCA engine's outcome to include the Neutron component, along with metrics related to VM launches and networking. The {*component, metrics list*} pairs with Neutron should be ranked higher than others.

To generate load on OpenStack, we run the 'boot_and_delete' task 100 times in the Rally benchmark [20], which launches 5 VMs concurrently and deletes them after 15-25 seconds. We apply this process to the correct (C) and faulty (F) versions of OpenStack. For the faulty version, the task fails as described above. We then apply the remaining stages of Sieve and feed the output to the RCA engine. For both versions, the dependency graphs are composed by 16 components, with 647 edges in the C version, and 343 edges in the F version. Below, we summarize the findings of RCA steps.

Steps #1 & #2: Metric analysis and component rankings. The total number of unchanged metrics exceeds that of 'novel' metrics (i.e., new and/or discarded) by ~ 4×. Furthermore, the initial component novelty ranking puts the Nova and Neutron components (known to be directly related with the anomaly) within the top 4 positions out of 16 (Table 5). This confirms the intuition behind our approach: novel metrics are more likely to be related to a failure.

Step #3: Cluster novelty & similarity. Computing the cluster novelty scores shows that the novel metrics from step 1 are distributed over only 27 of the 67 clusters (Figure 7(a)), even when conservatively considering a cluster to be novel if it contains at least one new or discarded metric. Considering only novel clusters reduces the number of metrics and the number of edges for the developers to analyze for the root cause in step 4. We also compute the similarity scores for these novel clusters and use the similarity in the next step.

Step #4: Edge filtering. By investigating the novel edges (i.e., new or deleted) in the dependency graph, the developers can better focus on understanding which component might be more relevant to the root cause. Utilizing different cluster similarity scores enables developers to filter out some of the edges that may not be relevant. Figures 7(b & c) show the effect of different cluster similarity thresholds for all components in Table 5 when filtering edges. Without any similarity thresholds, there are 42 edges of interest, corresponding to a set of 13 components, 29 clusters and 221 metrics that might be relevant to the root cause (Figure 7(c)). A higher threshold reduces the number of the {*component*, *metrics list*} pairs: filtering out clusters with inter-version similarity scores below 0.50, there are 24 edges of interest, corresponding to 10 components, 16 clusters and 163 metrics.

Figure 8 shows the edges between the components at the top-5 rows of Table 5, with a similarity threshold of 0.50. Note that one component (i.e., Nova scheduler) was removed by the similarity filter. Also, one of the new edges includes a Nova API component cluster, in which the *nova-instances-in-state-ACTIVE* metric is replaced with *nova-instances-in-state-ERROR*. This change relates directly to the observed anomaly (i.e., error in VM launch). The other end of this edge is a cluster in the Neutron component, which aggregates metrics related to VM networking, including a metric named *neutron-ports-in-status-DOWN*. This observation indicates a causal relationship between the VM failure and a VM networking issue, which is the true root cause of the anomaly.

We also note that a high similarity threshold may filter out useful information. For example, the Neutron component cluster with the *neutron-ports-in-status-DOWN* metric is removed with similarity thresholds above 0.60. We leave the study of this parameter's sensitivity to future work.

Step #5: Final rankings. The rightmost column on Table 5 shows the final rankings, considering edge filtering step with a 0.50 similarity threshold. Figure 8 shows a significant reduction in terms of state to analyze (from a total of 16 components and 508 metrics to 10 and 163, respectively) because of the exclusion of non-novel clusters. For example, for Nova API, the number of metrics reduces from 59 to 20 and for Neutron server from 42 to 22. Furthermore, our method includes the Neutron component as one of the top 5 components, and isolates an edge which is directly related with the true root cause of the anomaly.

7 Related Work

Scalable Monitoring. With the increasing number of metrics exposed by distributed cloud systems, the scalability of the monitoring process becomes crucial. Meng et al. [70] optimize monitoring scalability by choosing appropriate monitoring window lengths and adjusting the monitoring intensity at runtime. Canali et al. [39] achieve scalability by clustering metric data. A fuzzy logic approach is used to speed up clustering, and thus obtain data for decision making within shorter periods. Rodrigues et al. [43] explore the trade-off between timeliness and the scalability in cloud monitoring, and analyze the mutual influence between these two aspects based on the monitoring parameters. Our work is complementary to existing monitoring systems since Sieve aims to improve the efficiency by monitoring less number of metrics.

Distributed Debugging. Systems like Dapper [81] and Pip [77] require the developers to instrument the application to obtain its causal model. X-trace [47] uses a modified network stack to propagate useful information about the application. In contrast, Sieve does not modify the application code to obtain the call/dependency graph of the application.

Systems such as Fay [46] and DTrace [40] enable developers to dynamically inject debugging requests by developers and require no initial logs of metrics. Pivot Tracing [69] combines dynamic instrumentation with causality tracing. Sieve can complement these approaches, because it can provide information about interesting components and metrics, so that the developers can focus their efforts to understand them better. Furthermore, Sieve's dependency graph is a general tool that can not only be used for debugging, but also for other purposes such as orchestration [86–88].

Data provenance [34, 50, 83] is another technique that can be used to trace the dataflow in the system. Sieve can also leverage the existing provenance tools to derive the dependence graph.

Metric reduction. Reducing the size and dimensionality of the bulk of metric data exposed by complex distributed systems is essential for its understanding. Common techniques include sampling, and data clustering via k-means and k-medoids. Kollios et al. [64] employ biased sampling to capture the local density of datasets. Sampling based approaches argues for approximate computing [65, 74, 75] to enable a systematic trade-off between the accuracy, and efficiency to collect and compute on the metrics. Zhou et al. [91] simply use random sampling due to its simplicity and low complexity. Ng et al. [71] improved the k-medoid method and made it more effective and efficient. Ding et al. [44] rely on clustering over sampled data to reduce clustering time.

Sieve's approach is unique because of its two-step approach: (1) we first cluster time series to identify the internal dependency between *any* given metrics and then (2) infer the causal relations among time series. Essentially, Sieve uses two steps of data reduction for better reduction. Furthermore, Sieve's time series processing method extracts other useful information such as the time delay of the causal relationship, which can be leveraged in different use cases (e.g., root cause analysis).

Orchestration of autoscaling. Current techniques for autoscaling can be broadly classified into four categories [68]: *(i)* static and threshold-based rules (offered by most cloud computing providers [3, 4, 16, 25]); *(ii)* queuing theory [31, 57, 90]; *(iii)* reinforcement learning [76, 82, 89]; and *(iv)* time series analysis [41, 62, 79]. Existing systems using these techniques can benefit from the selection of better metrics and/or from the dependencies between components. In this regard, our work is complementary to these techniques: it is intended to provide the developers with knowledge about the application as a whole. In our case study, we showed the benefits of Sieve for an autoscaling engine using threshold-based rules.

Root Cause Analysis (RCA). Large and complex distributed systems are susceptible to anomalies, whose root causes are often hard

to diagnose [59]. Jiang et al. [61] compare "healthy" and "faulty" metric correlation maps, searching broken correlations. In contrast, Sieve leverages Granger causality instead of simple correlation, allowing for richer causality inference (e.g., causality direction, time lag between metrics). MonitorRank [63] uses metric collection for RCA in a service-oriented architecture. It only analyzes pre-established (component, metric) relations according to a previously-generated call graph. Sieve also uses a call graph, but does not fix metric relations between components, for a richer set of potential root causes. There are other application-specific solutions for RCA (e.g., Hansel [80], Gretel [52]). In contrast, Sieve uses a general approach for understanding the complexity of microservices-based applications that can support RCA as well as other use cases.

8 Experience and Lessons Learned

While developing Sieve, we set ourselves ambitious design goals (described in §2.2). However, we learned the following lessons while designing and deploying Sieve for real-world applications.

Lesson#1. When we first designed Sieve, we were envisioning a dependency graph that was clearly showing the relationships between components (e.g., a tree). As a result, not only would the number of metrics that needed to be monitored be reduced, but also the number of components: one would only need to observe the root(s) of the dependency graph, and make the actions of the dependent components according to the established relationships between the root(s) and them. Such a dependency graph would give the orchestration scenario a huge benefit. Unfortunately, our experience has shown us that the relationships between components are usually not linear, making the dependency graph more complex. Also, there was no obvious root. Consequently, we had to adjust our thinking and utilize some application knowledge regarding components and their relations with others. Nevertheless, in our experience, Sieve provides the developer with a good starting point to improve their workflows.

Lesson#2. Sieve is designed for "blackbox" monitoring of the evaluated application, where Sieve can collect and analyze generic system metrics that are exposed by the infrastructure (e.g., CPU usage, disk I/O, network bandwidth). However, in our experience, a system for monitoring and analyzing an application should also consider application-specific metrics (e.g., request latency, number of error messages) to build effective management tools. Fortunately, many microservices applications we analyzed already export such metrics. However, given the number of components and exported metrics, this fact can easily create an "information overload" for the application developers. In fact, the main motivation of Sieve was to deal with this "information overload". Our experience showed that Sieve can still monitor the application in the blackbox mode (i.e., no instrumentation to the application), but also overcome the barrage of application-specific metrics.

Lesson#3. To adapt to the application workload variations, Sieve needs to build a robust model for the evaluated application. This requires a workload generator that can stress-test the application thoroughly. To meet this requirement, there are three approaches: (1) In many cases the developers already supply an application-specific workload generator. For instance, we employed the workload generator shipped with the OpenStack distribution. (2) For cases where we did not have an existing workload generator, we implemented a custom workload generator for the evaluated application. For example, we built a workload generator for ShareLatex. Although we were able to faithfully simulate user actions in ShareLatex, such an approach might not be feasible for some applications. Having the ability to utilize existing production traces (e.g., by replaying the trace or by reproducing similar traces) or working in an online fashion to generate the model of the application would certainly help Sieve. Custom workload generation can then be used to close the gaps in the model for certain workload conditions not covered by the existing traces. (3) We could also explore some principled approaches for automatic workload generation, such as symbolic execution in distributed systems [33].

9 Conclusion and Future Work

This paper reports on our experiences with designing and building Sieve, a platform to automatically derive actionable insights from monitored metrics in distributed systems. Sieve achieves this goal by automatically reducing the amount of metrics and inferring inter-component dependencies. Our general approach is independent of the application, and can be deployed in an unsupervised mode without prior knowledge of the time series of metrics. We showed that Sieve's resulting model is consistent, and can be applied for common use cases such as autoscaling and root-cause debugging.

An interesting research challenge for the future would be to integrate Sieve into the continuous integration pipeline of an application development. In this scenario, the dependency graph can be updated incrementally [36–38], which would speed up the analytics part. In this way, the developers would be able to get real-time profile updates of their infrastructure. Another challenge is to utilize already existing traffic to generate the dependency graph without requiring the developers to load the system. Using existing traffic would alleviate the burden of developers to supply a workload generator. On the other hand, existing traffic traces might not always capture the stress points of the application. A hybrid approach, in which workload generation is only used for these corner cases, might help to overcome this problem.

Additional results and software availability. A detailed technical report with additional experimental evaluation results is available online [84]. Finally, the source code of Sieve is publicly available: https://sieve-microservices.github.io/.

Acknowledgments. We would like to thank Amazon AWS for providing the required infrastructure to run the experiments.

References

[1] Amazon AWS - Scaling Based on Metrics. https://docs.aws.amazon.com/autoscaling/latest/userguide/policy_creating.html. Last accessed: September, 2017.
[2] Amazon CloudWatch. https://aws.amazon.com/de/cloudwatch/. Last accessed: September, 2017.
[3] Amazon Web Services. https://aws.amazon.com/documentation/autoscaling/. Last accessed: September, 2017.
[4] Google Cloud Platform. https://cloud.google.com/developers/articles/auto-scaling-on-the-google-cloud-platform. Last accessed: September, 2017.
[5] Google Stackdriver. https://cloud.google.com/stackdriver/. Last accessed: September, 2017.
[6] Http Trace of WorldCup98. http://ita.ee.lbl.gov/html/contrib/WorldCup.html. Last accessed: September, 2017.
[7] InfluxDB. https://influxdata.com/time-series-platform/influxdb/. Last accessed: September, 2017.
[8] Introducing Vector. http://techblog.netflix.com/2015/04/introducing-vector-netflixs-on-host.html. Last accessed: September, 2017.
[9] Kapacitor. https://influxdata.com/time-series-platform/kapacitor/. Last accessed: September, 2017.
[10] Locust - An Open Source Load Testing Tool. http://locust.io/. Last accessed: September, 2017.

[11] Manualpage of Apache Benchmark. https://httpd.apache.org/docs/2.4/programs/ab.html. Last accessed: September, 2017.

[12] Microsoft Azure Monitor. https://docs.microsoft.com/en-us/azure/monitoring-and-diagnostics/monitoring-overview. Last accessed: September, 2017.

[13] Monitoring at Quantcast. https://www.quantcast.com/wp-content/uploads/2013/10/Wait-How-Many-Metrics_-Quantcast-2013.pdf. Last accessed: September, 2017.

[14] Nginx. https://nginx.org/. Last accessed: September, 2017.

[15] Observability at Uber Engineering: Past, Present, Future. https://www.youtube.com/watch?v=2JAnmzVwgP8. Last accessed: September, 2017.

[16] OpenStack. https://wiki.openstack.org/wiki/Heat. Last accessed: September, 2017.

[17] Openstack: API References (Response parameters). https://developer.openstack.org/api-ref/. Last accessed: September, 2017.

[18] Openstack: Open source Software for Creating Private and Public Clouds. https://www.openstack.org/. Last accessed: September, 2017.

[19] Openstack: Telemetry. https://docs.openstack.org/admin-guide/telemetry-measurements.html. Last accessed: September, 2017.

[20] Rally. https://wiki.openstack.org/wiki/Rally. Last accessed: September, 2017.

[21] Rancher Container Management. http://rancher.com/. Last accessed: September, 2017.

[22] Sharelatex - A Web-based Collaborative LaTeX Editor. https://sharelatex.com. Last accessed: September, 2017.

[23] Sysdig. http://www.sysdig.org/. Last accessed: September, 2017.

[24] Telegraf: Time-series Data Collection. https://www.influxdata.com/time-series-platform/telegraf/. Last accessed: September, 2017.

[25] Windows Azure. http://msdn.microsoft.com/en-us/library/hh680945(v=pandp.50).aspx. Last accessed: September, 2017.

[26] Openstack Kolla. http://docs.openstack.org/developer/kolla/, 2016. Last accessed: September, 2017.

[27] Openstack Kolla Launchpad: neutron-openvswitch-agent Bug. https://bugs.launchpad.net/kolla/+bug/1533942, 2016. Last accessed: September, 2017.

[28] *ptrace(2) Linux User's Manual*, 4.07 edition, Aug 2016.

[29] Scikit Documentation: sklearn.metrics.silhouette_score. http://scikit-learn.org/stable/modules/generated/sklearn.metrics.silhouette_score.html, 2016. Last accessed: September, 2017.

[30] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance Debugging for Distributed Systems of Black Boxes. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[31] A. Ali-Eldin, M. Kihl, J. Tordsson, and E. Elmroth. Efficient Provisioning of Bursty Scientific Workloads on the Cloud Using Adaptive Elasticity Control. In *Proceedings of the 3rd Workshop on Scientific Cloud Computing Date (ScienceCloud)*, 2012.

[32] Amemiya, Takeshi. *Advanced econometrics*. 1985.

[33] R. Banabic, G. Candea, and R. Guerraoui. Finding trojan message vulnerabilities in distributed systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[34] A. Bates, D. J. Tian, K. R. Butler, and T. Moyer. Trustworthy whole-system provenance for the linux kernel. In *24th USENIX Security Symposium (USENIX Security*, 2015.

[35] R. Bellman and R. Corporation. *Dynamic Programming*. Rand Corporation research study. Princeton University Press, 1957.

[36] P. Bhatotia. *Incremental Parallel and Distributed Systems*. PhD thesis, Max Planck Institute for Software Systems (MPI-SWS), 2015.

[37] P. Bhatotia, P. Fonseca, U. A. Acar, B. Brandenburg, and R. Rodrigues. iThreads: A Threading Library for Parallel Incremental Computation. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[38] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquini. Incoop: MapReduce for Incremental Computations. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2011.

[39] C. Canali and R. Lancellotti. An Adaptive Technique To Model Virtual Machine Behavior for Scalable Cloud Monitoring. In *Proceedings of the 19th IEEE Symposium on Computers and Communications (ISCC)*, 2014.

[40] B. Cantrill, M. W. Shapiro, A. H. Leventhal, et al. Dynamic instrumentation of production systems. In *Proceedings of the 2004 USENIX Annual Technical Conference (ATC)*, 2004.

[41] G. Chen, W. He, J. Liu, S. Nath, L. Rigas, L. Xiao, and F. Zhao. Energy-aware Server Provisioning and Load Dispatching for Connection-intensive Internet Services. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2008.

[42] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch. The Mystery Machine: End-to-end Performance Analysis of Large-scale Internet Services. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[43] G. da Cunha Rodrigues, R. N. Calheiros, M. B. de Carvalho, C. R. P. dos Santos, L. Z. Granville, L. Tarouco, and R. Buyya. The Interplay Between Timeliness and Scalability In Cloud Monitoring Systems. In *Proceedings of the 20nd IEEE Symposium on Computers and Communications (ISCC)*, 2015.

[44] R. Ding, Q. Wang, Y. Dang, Q. Fu, H. Zhang, and D. Zhang. Yading: Fast Clustering of Large-scale Time Series Data. In *Proceedings of the 41st International Conference on VERY LARGE DATA BASES (VLDB)*, 2015.

[45] C. W. S. Emmons, and B. Gregg. A Microscope on Microservices. http://techblog.netflix.com/2015/02/a-microscope-on-microservices.html, 2015. Last accessed: September, 2017.

[46] U. Erlingsson, M. Peinado, S. Peter, and M. Budiu. Fay: Extensible distributed tracing from kernels to clusters. In *Proceedings of the 23th ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[47] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-trace: A pervasive network tracing framework. In *Proceedings of the conference on Networked systems design & implementation (NSDI)*, 2007.

[48] M. Fowler. Microservices. http://martinfowler.com/articles/microservices.html. Last accessed: September, 2017.

[49] K. P. F.R.S. LIII. On Lines and Planes of Closest Fit to Systems of Points in Space. In *Proceedings of the 17th ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems (SIGMOD)*, 1901.

[50] A. Gehani and D. Tariq. Spade: Support for provenance auditing in distributed environments. In *Proceedings of the 13th International Middleware Conference (Middleware)*, 2012.

[51] D. Giles. Testing for Granger Causality. https://davegiles.blogspot.de/2011/04/testing-for-granger-causality.html. Last accessed: September, 2017.

[52] A. Goel, S. Kalra, and M. Dhawan. GRETEL: Lightweight Fault Localization for OpenStack. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*, 2016.

[53] C. Granger and P. Newbold. Spurious Regressions in Econometrics. *Journal of Econometrics*, 2(2):111–120, 1974.

[54] C. W. J. Granger. Investigating Causal Relations by Econometric Models and Cross-spectral Methods. *Econometrica*, 1969.

[55] W. H. Greene. *Econometric Analysis*. Prentice Hall, 5. edition, 2003.

[56] E. Haddad. Service-Oriented Architecture: Scaling the uber Engineering Codebase As We Grow. https://eng.uber.com/soa/, 2015. Last accessed: September, 2017.

[57] R. Han, M. M. Ghanem, L. Guo, Y. Guo, and M. Osmond. Enabling Cost-aware and Adaptive Elasticity of Multi-tier Cloud Applications. 2014.

[58] B. Harrington and R. Rapoport. Introducing Atlas: Netflix's Primary Telemetry Platform. http://techblog.netflix.com/2014/12/introducing-atlas-netflixs-primary.html, 2014. Last accessed: September, 2017.

[59] V. Heorhiadi, S. Rajagopalan, H. Jamjoom, M. K. Reiter, and V. Sekar. Gremlin: Systematic Resilience Testing of Microservices. In *Proceedings of the 2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, 2016.

[60] M. A. Jaro. Advances in record-linkage methodology as applied to matching the 1985 census of tampa, florida. 84(406):414–420, 1989.

[61] M. Jiang, M. A. Munawar, T. Reidemeister, and P. A. S. Ward. Dependency-aware Fault Diagnosis with Metric-correlation Models in Enterprise Software Systems. In *Proceedings of 2010 International Conference on Network and Service Management (NSDI)*, 2010.

[62] S. Khatua, A. Ghosh, and N. Mukherjee. Optimizing the Utilization of Virtual Resources in Cloud Environment. In *Proceedings of the 2010 IEEE International Conference on Virtual Environments, Human-Computer Interfaces and Measurement Systems (CIVEMSA)*, 2010.

[63] M. Kim, R. Sumbaly, and S. Shah. Root Cause Detection in a Service-oriented Architecture. In *Proceedings of the ACM SIGMETRICS/international conference on Measurement and modeling of computer systems (SIGMETRICS)*, 2013.

[64] G. Kollios, D. Gunopulos, N. Koudas, and S. Berchtold. Efficient Biased Sampling for Approximate Clustering and Outlier Detection in Large Data Sets. In *Proceedings of the 2003 IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 2003.

[65] D. R. Krishnan, D. L. Quoc, P. Bhatotia, C. Fetzer, and R. Rodrigues. IncApprox: A Data Analytics System for Incremental Approximate Computing. In *proceedings of International Conference on World Wide Web (WWW)*, 2016.

[66] J. M. Liu. Nonlinear Time Series Modeling Using Spline-based Nonparametric Models. In *Proceedings of the 15th American Conference on Applied Mathematics (AMATH)*, 2009.

[67] R. Lomax and D. Hahs-Vaughn. *Statistical Concepts: A Second Course, Fourth Edition*. Taylor & Francis, 2012.

[68] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano. A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments. In *Proceedings of Grid Computing (CCGrid)*, 2014.

[69] J. Mace, R. Roelke, and R. Fonseca. Pivot tracing: Dynamic causal monitoring for distributed systems. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*, 2015.

[70] S. Meng and L. Liu. Enhanced Monitoring-as-a-service for Effective Cloud Management. In *Proceedings of IEEE Transactions on Computers (TC)*, 2013.

[71] R. T. Ng and J. Han. Efficient and Effective Clustering Methods for Spatial Data Mining. In *Proceedings of the 19st International Conference on VERY LARGE DATA BASES (VLDB)*, 1994.

[72] C. H. Papadimitriou, P. Raghavan, H. Tamaki, and S. Vempala. Latent semantic indexing: A probabilistic analysis. *Journal of Computer and System Sciences*, 2000.

[73] J. Paparrizos and L. Gravano. k-Shape: Efficient and Accurate Clustering of Time Series. In *Proceedings Of the 2016 ACM SIGMOD/PODS Conference (SIGMOD)*,

2016.

[74] D. L. Quoc, M. Beck, P. Bhatotia, R. Chen, C. Fetzer, and T. Strufe. PrivApprox: Privacy-Preserving Stream Analytics. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC)*, 2017.

[75] D. L. Quoc, R. Chen, P. Bhatotia, C. Fetzer, V. Hilt, and T. Strufe. StreamApprox: Approximate Computing for Stream Analytics. In *Proceedings of the International Middleware Conference (Middleware)*, 2017.

[76] J. Rao, X. Bu, C.-Z. Xu, L. Wang, and G. Yin. VCONF: A Reinforcement Learning Approach to Virtual Machines Auto-configuration. In *Proceedings of the 6th International Conference on Autonomic Computing (ICAC)*, 2009.

[77] P. Reynolds, C. E. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: Detecting the unexpected in distributed systems. In *Proceedings of the conference on Networked systems design & implementation (NSDI)*, 2006.

[78] P. J. Rousseeuw. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics*, 20:53 – 65, 1987.

[79] N. Roy, A. Dubey, and A. Gokhale. Efficient Autoscaling in the Cloud Using Predictive Models for Workload Forecasting. In *Proceedings of the 2011 IEEE 4th International Conference on Cloud Computing (CCIS)*, 2011.

[80] D. Sharma, R. Poddar, K. Mahajan, M. Dhawan, and V. Mann. Hansel: Diagnosing Faults in OpenStack. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2015.

[81] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. Technical report, Google, 2010.

[82] G. Tesauro, N. K. Jong, R. Das, and M. N. Bennani. A Hybrid Reinforcement Learning Approach to Autonomic Resource Allocation. In *Proceedings of the 2006 IEEE International Conference on Autonomic Computing (ICAC)*, 2006.

[83] J. Thalheim, P. Bhatotia, and C. Fetzer. Inspector: Data Provenance using Intel Processor Trace (PT). In *proceedings of IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2016.

[84] J. Thalheim, A. Rodrigues, I. E. Akkus, P. Bhatotia, R. Chen, B. Viswanath, L. Jiao, and C. Fetzer. Sieve: Actionable Insights from Monitored Metrics in Microservices. *ArXiv e-prints*, Sept. 2017.

[85] N. X. Vinh, J. Epps, and J. Bailey. Information Theoretic Measures for Clusterings Comparison: Is a Correction for Chance Necessary? In *Proceedings of the 26th Annual International Conference on Machine Learning (ICML)*, 2009.

[86] A. Wieder, P. Bhatotia, A. Post, and R. Rodrigues. Brief Announcement: Modelling MapReduce for Optimal Execution in the Cloud. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of Distributed Computing (PODC)*, 2010.

[87] A. Wieder, P. Bhatotia, A. Post, and R. Rodrigues. Conductor: Orchestrating the Clouds. In *Proceedings of the 4th international workshop on Large Scale Distributed Systems and Middleware (LADIS)*, 2010.

[88] A. Wieder, P. Bhatotia, A. Post, and R. Rodrigues. Orchestrating the Deployment of Computations in the Cloud with Conductor. In *Proceedings of the 9th USENIX symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[89] L. Yazdanov and C. Fetzer. Lightweight Automatic Resource Scaling for Multi-tier Web Applications. In *Proceedings of the 2014 IEEE 7th International Conference on Cloud Computing (CLOUD)*, 2014.

[90] Q. Zhang, L. Cherkasova, and E. Smirni. A Regression-Based Analytic Model for Dynamic Resource Provisioning of Multi-Tier Applications. In *Proceedings of the Fourth International Conference on Autonomic Computing (ICAC)*, 2007.

[91] S. Zhou, A. Zhou, J. Cao, J. Wen, Y. Fan, and Y. Hu. Combining Sampling Technique With DBSCAN Algorithm for Clustering Large Spatial Databases. In *Proceedings of the 4th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD)*, 2000.